

INSTANT

Short | Fast | Focused

Vert.x

A kick-start for rapid development of asynchronous network applications using the Vert.x framework

Simone Scarduzio

[PACKT] PUBLISHING

Table of Contents

Java API 版本的Vert.x Core 手册

- 欢迎关注http://quanke.name/
- 交流群： `231419585`
- 阅读地址：http://vertx.help/
- 下载地址：https://www.gitbook.com/book/quanke/vert-x-core-manual-for-java
- 本书源码地址：https://github.com/quanke/vert-x-core-manual-for-java

本人英语水平有限，有任何问题，请加群交流 ： `231419585`

源码在github上

Vert.x Core提供的功能：

- 编写TCP客户端和服务器
- 编写 HTTP 客户端和服务器包括 Websocket 支持
- 事件总线(Event bus)
- 共享的数据-本地的map和分布式的map
- 定时和延时运行
- 部署和非部署 Verticles
- Sockets
- DNS 客户端
- 文件系统
- 高可用性
- 集群

Vert.x核心功能是相当简单的 —— 你不会找到数据库访问、 授权或高级别 web 功能等，这些东西你可以在哪里找到？在这里-，**Vert.x ext**(扩展)。

Vert.x core 非常小，非常轻量级。只是使用你想要的部分。也是完全可嵌入在您现有的应用程序 —— 不强迫你使用特殊方式架构您的应用程序，这样你可以方向使用 Vert.x。

您可以使用任何 Vert.x 支持的其他语言的核心。这有点小酷-我们不强迫你使用 Java API ，JavaScript 或者 Ruby等都没问题 —— 毕竟，不同的语言有不同的习惯和语法，迫使Ruby开发人员使用 Java 的语法，这会很奇怪 (举个例子)。相反，我们自动生成以 Java Api 为核心，等效、地道的每种语言。

从现在起我们会使用 core 指 Vert.x core。

如果你使用 Maven 或 Gradle，需要增加以下依赖才能使用Vert.x Core API:

- Maven (在你的pom.xml中增加):

```
<dependency>
  <groupId>io.vertx</groupId>
  <artifactId>vertx-core</artifactId>
  <version>3.2.0</version>
</dependency>
```

- Gradle (在您的build.gradle文件增加):

```
compile io.vertx:vertx-core:3.2.0
```

下面让我们来讨论 `core` 的不同概念和功能。

从Vert.x开始

注意：这大部分是*Java*特有的-需要语言特有的调用方法

如果没有获得Vertx对象，Vert.x做不了什么。

Vertx对象是 Vert.x 的控制中心，几乎可以做所有事，包括创建客户端和服务器，获取引用到事件总线（event bus）、设置计时器等。

所以怎么获得Vertx实例?

如果已经嵌入了 Vert.x，然后只需创建一个实例，如下所示:

```
Vertx vertx = Vertx.vertx();
```

如果使用 Verticles

注意*:*大多数应用程序只需要一个单一的 *Vert.x* 实例，但如果你需要，可以创建多个 *Vert.x* 实例，例如，事件总线或不同的服务器和客户端之间的隔离。

创建一个指定选项的Vertx 对象

创建一个 Vertx 对象时，如果默认值不是正确的选择，你还可以指定选项:

```
Vertx vertx = Vertx.vertx(new VertxOptions().setWorkerPoolSize(40));
```

VertxOptions对象有许多设置，可以配置集群、 高可用性、 池的大小等。所有设置细节在 Javadoc 中有描述。

创建群集 Vert.x 对象

如果您正在创建clustered（群集） Vert.x (更多集群相关的请参阅事件总线（ event bus）)，然后通常会使用异步方式创建 Vertx 对象。

这是因为不同的 Vert.x 实例在群集中组合在一起，通常需要一些时间 (也许几秒钟) 的。在这段时间，我们不想阻止调用线程，所以我们把结果以异步方式给你。

你是傻瓜吗？

你可能注意到，在前面使用fluent API（fluent API：流API，更易使用的API，也称傻瓜式API）的例子。

fluent API 是支持链式调用的。例如:

```
request.response().putHeader("Content-Type", "text/plain").write("some text").end();
```

整个 Vert.x Api都是这种模式，，所以要去适应它。

可以链式编写代码，当然你也可以按自己的喜欢，写上这样的代码：

```
HttpServerResponse response = request.response();
response.putHeader("Content-Type", "text/plain");
response.write("some text");
response.end();
```

不要call（调用、打电话）我们，我们会call给你。

Vert.x Api 是很大程度上由事件驱动的。这意味着，当事情发生在你感兴趣的Vert.x，Vert.x会通过回调方式向您发送events。

一些示例events:

- 计时器激活
- socket收到数据
- 从磁盘读取数据
- 发生了异常
- HTTP 服务器收到请求

通过向 Vert.x Api 提供处理程序来处理事件。例如要接收一个计时器事件每一秒你会做:

```
vertx.setPeriodic(1000, id -> {
  // This handler will get called every second
  System.out.println("timer fired!");
});
```

或接收到 HTTP 请求:

```
server.requestHandler(request -> {
  // This handler will be called every time an HTTP request is received at the server
  request.response().end("hello world!");
});
```

一段时间后当 Vert.x 有一个事件，它将传递到您的处理程序 Vert.x 将它异步调用.

这将引导我们进入Vert.x 中的一些重要概念:

不要阻塞我!

除了极少数例外 (一些文件系统操作的“同步”结束)，没有一个 Vert.x Api 阻塞调用线程。

如果可以立即提供的结果，它将立即返回，你通常会提供一个handle来接收过一段时间的事件。

由于Vert.x API没有任何阻塞的线程，这意味着你可以使用Vert.x来处理只是使用小数目线程的大量并发。

常规阻塞API使用线程可能会阻塞：

- 从socket读取数据
- 向磁盘写入数据
- 向收件人发送一条消息，等待答复。
- …

在所有上述情况下，当您的线程正在等待结果时它不能做别的-这是实际上是浪费。

这意味着，如果你需要大量的并发使用阻塞 APIs，然后你需要大量的线程，以防止您的应用程序停止工作。

线程在他们所需要的内存（例如栈）和上下文切换方面有开销。

对于许多现代应用程序所需要的并发水平，阻塞的方法不能按比例缩放。

Reactor和多Reactor

之前提到Vert.x API是事件驱动 - 当他们都可用时，Vert.x传递事件给处理程序。

在大多数情况下Vertx要求使用一种称为event loop线程的处理程序。

如无有 Vert.x 或您的应用程序块中，event loop可以欢快地运行将事件传递给不同的处理程序提供事件陆续到达。

因为没有阻塞，event loop可以在短时间内提供大量的事件。例如一个单一的event loop可以非常迅速地处理成千上万的 HTTP 请求。

我们把这个叫做反应器模式（Reactor Pattern）.

你可能会有之前听说过-例如 Node.js 实现此模式。

标准的Reactor所有事件都运行在单一事件循环线程。

单个线程的麻烦是在任何一个时间它只能运行在单一的核心上(例如 Node.js 应用，如果想要实现多线程你要做很多事 。

而Vert.x 不同。不是单事件循环，每个 Vertx 实例都维护若干个事件循环。默认情况下，我们选择数量基于在机器上可用的内核数，但可以自己设置。

与 Node.js 不同是Vertx进程是可配置的，与 Node.js 不同

我们称这种模式多反应器（Multi-Reactor）模式，以区别于单线程的反应器模式。

注意：即使 *Vertx* 实例维护多个事件循环，任何特定的处理程序将永远不会被同时执行，在大多数情况下 *(*除了 *worker verticles)* 将始终使用完全相同的事件循环调用。

黄金法则 — 不要阻塞事件循环

我们已经知道 Vert.x Api 是非阻塞，并且不会堵塞事件循环。 如果你堵塞事件循环，那事件循环将不能做别的事，因为它被阻塞了。如果所有的event loop被阻塞了，应用程序将完全停止！

所以不要这样做!你已经被警告。

阻塞的例子包括:

- Thread.sleep()
- 等待锁
- 等待互斥体或监视器 (例如同步段)
- 做一个长时间的数据库操作和等待返回
- 做复杂的计算，需要很长的时间。
- 死循环。

如果有上述情况停止了事件循环（event loop），需要相当长的时间，你应经立即去下一步，并等待进一步的指示。

这个时间具体多长？

具体多长时间?它取决于应用程序需要的并发量。

如果你有一个单一的事件循环，并且你想要处理每秒 10000 的 http 请求，然后很明显，每个请求不能超过 0.1 ms 要处理，所以你不能阻塞比这更多的时间。

这道数学题并不是困难，作为练习留给读者。

如果您的应用程序不响应，可能你阻塞的事件循环的地方。为了帮助您诊断此类问题，如果它检测到一段时间后事件循环还没有恢复，Vert.x会自动记录警告。如果你在日志中看到这样的警告，那么你就应该去检查应用。

```
Thread vertx-eventloop-thread-3 has been blocked for 20458 ms
```

Vert.x 还将提供 `堆栈跟踪` 来确定阻塞发生的位置。

如果你想关闭这些警告或更改设置，你可以在创建Vertx对象之前，使用VertxOptions配置。

运行阻塞代码

在完美的世界，将没有战争或饥饿，所有 Api 将使用异步写，阳光明媚，绿色的草地有跳来跳去的兔子和手牵手的小羊羔。

但是，现实世界并不是这样。(你看过新闻最近吗?)

事实是，大多数库，特别是在JVM的生态,Y有许多是同步API，许多的方法有可能阻塞。一个很好的例子是JDBC API - 这是本质上的同步，不管如何努力尝试，Vert.x 不能撒上魔法使之同步。

我们不打算在一夜之间把一切改写成异步，所以我们需要给你提供一个方法，一个Vert.x应用中安全地使用“传统”的阻塞API的方法。

如前所述，直接在事件循环里调用阻塞操作，会妨碍它做任何其他有用的工作。所以你怎么能这样呢?

它是通过调用 `executeBlocking` 指定要执行的阻塞的代码和在执行阻塞的代码时调用返回异步结果处理程序。

通过调用 `executeblocking` ，执行阻塞代码，当阻塞代码执行完成后通过异步回调的方式返回

```
vertx.executeBlocking(future -> {
  // Call some blocking API that takes a significant amount of time to return
  String result = someAPI.blockingMethod("hello");
  future.complete(result);
}, res -> {
  System.out.println("The result is: " + res.result());
});
```

默认情况下，如果 `executeBlocking` 从相同的上下文 (例如同一垂直实例) 调用几次不同的 executeBlocking 则以串行方式执行 (即一个接一个)。

默认情况下，如果 `executeBlocking` 在同一环境（例如同一个verticle实例）多次调用，那么不同的 `executeBlocking` 将串行执行（即一个接一个）。

如果不关系执行顺序，调用 `executeBlocking` 时可以制定 `ordered` 参数为 `false` 。在这种情况下 `executeBlocking` 会与worker pool并行执行。

运行阻塞的代码替代方法是使用worker verticle

worker verticle始终在worker池中的线程执行。

Verticles

Vert.x 带有一个简单、 可扩展，actor-like 开箱即用，你可以用来节省您编写您自己的部署和并发模型。

此模型是完全可选的, 如果你不想去用，Vert.x 不会强制您以这种方式创建应用程序。

该模型并不要求是一个严格的actor-model 的实现，但它确实有相似之处，特别是对并发性，扩展和部署。

若要使用此模型，把代码设置为**verticles**.

Verticles 是代码的得到部署和运行的 Vert.x 块。Verticles 可以使用任何 Vert.x 支持的语言编写，单个应用程序包含 verticles， 可以使用多种语言编写。

你可能会想，verticle有点像 Actor Model 中的actor。

应用通常是同一个 Vert.x 实例，同时由多个verticle实例组成。不同的verticle实例通过event bus发送消息。

编写 Verticles

Verticle类必须实现Verticle接口。

如果喜欢可以直接实现Verticle接口，但是通常简答的方法是继承抽象类AbstractVerticle

下面是Verticles示例:

```
public class MyVerticle extends AbstractVerticle {

  // Called when verticle is deployed
  public void start() {
  }

  // Optional - called when verticle is undeployed
  public void stop() {
  }

}
```

通常你会像在上面的示例一样重写 start 方法。

当 Vert.x 部署verticle后，会调用 start 方法，当调用completed后，说明verticle启动完毕。

您也可以选择可以覆盖 stop 方法。取消部署的时候会调用。

异步Verticle启动和停止

实现异步启动

下面是一个示例:

```
public class MyVerticle extends AbstractVerticle {

  public void start(Future<Void> startFuture) {
    // Now deploy some other verticle:

    vertx.deployVerticle("com.foo.OtherVerticle", res -> {
      if (res.succeeded()) {
        startFuture.complete();
      } else {
        startFuture.fail();
      }
    });
  }
}
```

同样地，也是 stop 方法也是异步。如果你想要做一些verticle的清理工作，这需要一些时间，则如此使用。

```
public class MyVerticle extends AbstractVerticle {

  public void start() {
    // Do something
  }

  public void stop(Future<Void> stopFuture) {
    obj.doSomethingThatTakesTime(res -> {
      if (res.succeeded()) {
        stopFuture.complete();
      } else {
        stopFuture.fail();
      }
    });
  }
}
```

信息: 你不需要在启动一个verticle后手动取消部署子 verticles，在verticle的 stop 方法。当父 verticle's取消部署时，Vert.x 将自动取消部署任何子 verticles。

2016-09-06 03:18:18

Verticle类型

有三种不同类型的 verticles:

标准 **Verticles**

这些都是最常见和最有用的类型 —— 他们总是使用事件循环线程执行。更多讨论在下一节。

Worker Verticles

一个实例是永远不会有多个线程并发执行。

多线程的**worker verticles**

一个实例可以由多个线程同时执行。

标准 verticles

标准verticles当创建和调用start方法时分配一个event loop。调用执行都在相同的event loop上。

这意味着我们可以保证您的verticles实例中的所有代码总是都执行相同的事件循环上 (只要你不调用它自己创建的线程!)。

这意味着可以在程序里作为单线程编写所有的代码，把担心线程和扩展的问题交给Vert.x。没有更多令人担忧的同步和更多不稳定的问题，也避免了多线程死锁的问题。

Worker verticles

Worker verticles就像标准的verticles一样，但不使用事件循环执行，从 Vert.x worker线程池使用一个线程。

worker verticles 专为调用阻塞的代码，因为他们不会阻止任何事件循环。

如果你不想使用worker verticles运行阻塞的代码，可以在事件循环上直接运行内联阻塞代码。

如果您要以worker verticles的方式部署verticle，需要调用 setWorker.

```
DeploymentOptions options = new DeploymentOptions().setWorker(true);
vertx.deployVerticle("com.mycompany.MyOrderProcessorVerticle", options);
```

Worker verticle实例永远不会有多个线程并发执行 ，但可以在不同的时间由不同的线程执行。

多线程Worker verticles

多线程的worker verticle就像正常worker verticle，但它是可以由不同的线程同时执行。

警告！ 多线程的*worker verticle* 是一项高级的功能，大多数应用程序会对他们来说没有必要。因为在这些 *verticles* 并发，你必须非常小心，使用标准的 *Java* 技术的多线程编程，以保持*verticle*一致状态。

以编程方式部署 verticles

使用deployVerticle方法部署verticles，指定verticles名，也可以把已经创建的verticles实例传递过来。

注意！ 部署*verticles*实例只是 *Java*。

```
Verticle myVerticle = new MyVerticle();
vertx.deployVerticle(myVerticle);
```

您还可以通过指定verticles名称部署 verticles.

verticles名称用于查找特定的VerticleFactory ，将用来实例化实际verticles实例。

不同verticle工厂可用于实例化不同语言verticles和其他各种原因，例如装载服务和从Maven在运行时得到verticles。

这允许您从 Vert.x 支持的任何语言编写的其他语言verticles部署 。

这里是 verticles 的部署一些不同类型的示例:

```
vertx.deployVerticle("com.mycompany.MyOrderProcessorVerticle");

// Deploy a JavaScript verticle
vertx.deployVerticle("verticles/myverticle.js");

// Deploy a Ruby verticle verticle
vertx.deployVerticle("verticles/my_verticle.rb");
```

verticle名称映射到一个verticle工厂的规则

当部署 verticle(s) 使用一个名称，该名称用于选择将实例化 verticle(s) 的实际verticle工厂。

verticle的名称可以有前缀-它是一个字符串, 后跟一个冒号，它如果存在，则会被用来查找工厂，如

```
js:foo.js // Use the JavaScript verticle factory
groovy:com.mycompany.SomeGroovyCompiledVerticle // Use the Groovy verticle factory
service:com.mycompany:myorderservice // Uses the service verticle factory
```

如果没有前缀，则 Vert.x 将寻找一个后缀和使用，例如查找工厂，

```
foo.js // Will also use the JavaScript verticle factory
SomeScript.groovy // Will use the Groovy verticle factory
```

如果没有前缀或后缀，则 Vert.x 会假设它是 Java 完整类名称 (FQCN)，然后实例化。

怎么样找到Verticle Factories?

大多数的Verticle factories在 Vert.x 启动时从类路径中加载并注册。

你可以通过编程方式注册和注销Verticle factories，使用registerVerticleFactory和unregisterVerticleFactory。

等待部署完成

Verticle部署是异步的，可能部署完成后才返回。

如果你想要部署完成后通知，您可以部署指定完成处理程序:

```
vertx.deployVerticle("com.mycompany.MyOrderProcessorVerticle", res -> {
  if (res.succeeded()) {
    System.out.println("Deployment id is: " + res.result());
  } else {
    System.out.println("Deployment failed!");
  }
});
```

如果部署成功，该完成包含部署 ID 字符串的结果传递给处理程序。

如果你想要取消部署，可以稍后使用此部署 ID。

取消 verticle 部署

部署可以通过undeploy取消部署.

取消部署本身是异步的，所以如果想要在取消部署完成后通知，您可以部署指定完成处理程序:

```
vertx.undeploy(deploymentID, res -> {
  if (res.succeeded()) {
    System.out.println("Undeployed ok");
  } else {
    System.out.println("Undeploy failed!");
  }
});
```

指定verticle实例数

在部署时verticle使用verticle的名称，可以指定您要部署的verticle实例的数目:

```
DeploymentOptions options = new DeploymentOptions().setInstances(16);
vertx.deployVerticle("com.mycompany.MyOrderProcessorVerticle", options);
```

用于跨多个内核轻松扩展。例如，您可能有 web 服务器verticle部署，服务器是多核的，你想要部署多个实例来充分利用所有核心。

配置verticle

在部署的时可以传递一个JSON形式的配置给verticle：

```
JsonObject config = new JsonObject().put("name", "tim").put("directory", "/blah");
DeploymentOptions options = new DeploymentOptions().setConfig(config);
vertx.deployVerticle("com.mycompany.MyOrderProcessorVerticle", options);
```

这种配置可通过Context对象获得或直接使用config方法。

作为一个 JSON 对象返回的配置，您可以检索数据，如下所示:

```
System.out.println("Configuration: " + config().getString("name"));
```

在Verticle里访问环境变量。

使用 Java API 访问环境变量和系统属性:

```
System.getProperty("prop");
System.getenv("HOME");
```

Verticle隔离组

默认情况下，Vert.x 具有flat classpath。即，当 Vert.x 部署 verticles 使用当前类加载器-它不会创建一个新。在大多数情况下这是最简单、 最明确和理智的事情。

然而，在某些情况下，您可能想要部署verticle，所以在您的应用程序verticle的类是孤立于其他。

这可能并非如此，例如，如果您要部署两个不同版本的同一个 Vert.x 实例，类名相同的verticle或如果你有两个不同的 verticles，使用不同版本的相同的 jar 库。

隔离组，使用setisolatedclasses - 条目名称可以是完整类名，如 `com.mycompany.myproject.engine.myclass` 或它可以是一个通配符，将匹配任何包中的类和任何子的软件包，例如 `com.mycompany.myproject.*` 将匹配 `com.mycompany.myproject` 包中的任何类或任何子包。

请注意，只有匹配的将隔离 —— 任何其他类将由当前的类加载器加载。

也可以与setExtraClasspath提供附加的类路径条目，所以如果你想要加载的类或资源，已经不是目前的主要的类路径上你可以添加这。

如果你想加载尚不存在的主要类路径类或资源，你可以使用setExtraClasspath

警告! 谨慎使用此功能。类加载程序是一罐蠕虫，增加调试困难，除其他事项外。

这里是一个使用隔离组隔离verticle deployment的示例。

```
DeploymentOptions options = new DeploymentOptions().setIsolationGroup("mygroup");
options.setIsolatedClasses(Arrays.asList("com.mycompany.myverticle.*",
                   "com.mycompany.somepkg.SomeClass", "org.somelibrary.*"));
vertx.deployVerticle("com.mycompany.myverticle.VerticleClass", options);
```

高可用性（High Availability）

高可用性 (HA)可以在Verticles deployed时启动，当vert.x 的实例突然死了，从集群重新部署另外的vert.x 实例。

若要启用高可用性运行verticle，只是追加 `-ha` 开关:

```
vertx run my-verticle.js -ha
```

当启用高可用性，无需添加 `-cluster` 。

有关高可用性功能和配置的高可用性和故障转移（High Availability and Fail-Over）部分，有更多细节。

从命令行运行 Verticles

使用 Vert.x ，通常可以直接在 Maven 或 Gradle 项目中添加 Vert.x core 库依赖。

还可以直接从命令行运行 Vert.x verticles。

做到这一点，你需要下载和安装一个 Vert.x ，并将安装的 `bin` 目录添加到 `PATH` 环境变量。还要确保 `PATH` 有 `Java 8 JDK` .

注意*! JDK*是需要支持的*Java*代码的即时编译。

现在可以通过使用 `vertx run` 命令运行 verticles。这里有一些例子:

```
# Run a JavaScript verticle
vertx run my_verticle.js

# Run a Ruby verticle
vertx run a_n_other_verticle.rb

# Run a Groovy script verticle, clustered
vertx run FooVerticle.groovy -cluster
```

你甚至可以不编译直接运行Java源代码！

```
vertx run SomeJavaSourceFile.java
```

Vert.x 在运行之前会先编译Java源文件。这可以快速原型开发verticles，不需要设置 Maven 或 Gradle !

更多信息，请在命令行上执行 `vertx` 。

Vert.x 退出

由 Vert.x 实例维护的线程不是守护程序线程，从而防止 JVM 退出。

如果你嵌入 Vert.x 和你已完成了，你可以调用close来关闭它。

这将关闭所有的内部线程池和关闭其他的资源，让 JVM 退出。

Context对象

当Vert.x提供一个事件的处理程序或调用Verticle的开始或停止的方法，执行与 `Context` 相关联。一般Context是event-loop context 绑定特定的事件循环线程。因此，对于这方面的执行总是发生在该完全相同的事件循环线程。在worker verticles和运行内嵌阻止代码worker context的情况下将使用一个线程从worker线程池的执行关联。

若要获得context，请使用getOrCreateContext方法:

```
Context context = vertx.getOrCreateContext();
```

如果当前线程具有一个与它相关联的context，它重复使用context对象。如果不创建新实例的context。您可以测试您取得的context的类型:

```
Context context = vertx.getOrCreateContext();
if (context.isEventLoopContext()) {
  System.out.println("Context attached to Event Loop");
} else if (context.isWorkerContext()) {
  System.out.println("Context attached to Worker Thread");
} else if (context.isMultiThreadedWorkerContext()) {
  System.out.println("Context attached to Worker Thread - multi threaded worker");
} else if (! Context.isOnVertxThread()) {
  System.out.println("Context not attached to a thread managed by vert.x");
}
```

当您取得context对象时，您可以以异步方式在此context中运行代码。换句话说，您提交相同的context，但后来将最终运行任务:

```
vertx.getOrCreateContext().runOnContext( (v) -> {
  System.out.println("This will be executed asynchronously in the same context");
});
```

当几个处理程序在相同的context中运行时，他们可能想要分享数据。context对象提供方法来存储和获取在context中共享的数据。例如，它可以让你通过一些行动runOnContext运行数据：

```
final Context context = vertx.getOrCreateContext();
context.put("data", "hello");
context.runOnContext((v) -> {
  String hello = context.get("data");
});
```

context对象还可让您使用的config方法访问verticle配置。

执行定期和延迟的操作

在Vert.x 中执行定期和延迟的操作是非常常见的。

在标准 verticles 中，不能使用thread sleep 引入延迟，这样会止事件循环线程。

相反，您可以使用 Vert.x 计时器。定时器可以一次性的计时器或定期的计时器。我们将讨论两个

一次性的计时器

一个单次定时器有一定的延迟之后调用一个事件处理程序，以毫秒为单位表示。

使用setTimeout方法启动计时器，

```
long timerID = vertx.setTimer(1000, id -> {
  System.out.println("And one second later this is printed");
});

System.out.println("First this is printed");
```

返回值是一个唯一的定时器 id，以后可用于取消计时器。该处理器还通过定时器id。

定期计时器

您还可以设置一个计时器来定期启动，通过使用setPeriodic方法。

会有一个初始延迟等于周期。

setPeriodic的返回值是一个唯一的计时器的 id (long)。如果以后计时器需要取消，可以使用 id。

传递到计时器事件处理程序的参数也是唯一的计时器的 id:

请记住，计时器会定期触发。如果你的周期性处理需要相当长的时间进行，你的计时器事件可以运行连续或更糟的是: 堆积。

在这种情况下，您应该考虑使用setTimer替代。一旦您处理已完成，您可以设置下一个计时器。

```
long timerID = vertx.setPeriodic(1000, id -> {
  System.out.println("And every second this is printed");
});

System.out.println("First this is printed");
```

取消计时器

若要取消一个定期的计时器，请调用cancelTimer指定的计时器的 id。例如:

```
vertx.cancelTimer(timerID);
```

Verticles 自动清理

如果您正在从 verticles 内创建的计时器，这些计时器将被自动关闭verticle undeployed。

事件总线（Event Bus）

event bus是Vert.x 的中枢神经系统 。

通过Vert.x实例使用eventBus方法得到单一的event bus实例。 事件总线允许您的应用程序相互沟通，不论何种语言，他们写的以及他们是否在同一个 Vert.x 实例，或在一个不同的 Vert.x 实例的不同部分。

它甚至可以弥合，允许客户端 JavaScript 运行在浏览器上相同的事件总线进行通信。

事件总线构成了一个分布式对等消息传递系统跨越多个服务器节点和多个浏览器。

事件总线支持发布/订阅，点到点和请求-响应消息。

事件总线 API 是非常简单的。它基本上涉及注册处理程序，注销处理程序和发送和发布消息。

第一次一些理论:

理论

处理

事件到地址总线上发送邮件.

Vert.x 并不费心处理计划任何幻想。在 Vert.x 的地址是只是一个字符串。任何字符串是有效的。然而它是计划的明智地使用某种类型，例如使用句点划定一个命名空间。

一些例子的有效地址是 europe.news.feed1、 acme.games.pacman、 香肠和 X。

处理程序

在处理程序接收消息。你注册一个处理程序的地址。

很多不同的处理程序可以注册在相同的地址。

单个处理程序可以注册在许多不同的地址。

发布/订阅消息传递

事件总线支持发布消息。

消息发布到的地址。发布是指信息传递给所有处理程序，注册在该地址。

这是熟悉的发布订阅消息传递模式。

点到点和请求-响应消息

事件总线还支持点到点消息传递。

邮件发送到某个地址。Vert.x 然后将传送到位于该地址注册的处理程序之一。

如果有多个处理程序，登记的地址，将选择一个使用非严格-轮转调度算法。

使用点到点消息传递，发送消息时，可以指定一个可选的答复处理程序。

当消息由收件人收到，并且已被处理时，收件人可以选择决定对消息进行回复。如果他们这样做答复处理程序将被调用。

当回到在发件人收到的答复时，它也可以被回复。这可以是重复的广告-无穷尽，并允许一个对话框，可以设置两个不同的 verticles 之间。

这是一种常见的消息传递模式，称为请求-响应模式。

尽最大努力交付

Vert.x 最好来传递邮件，并不会有意识地把它们扔掉。这就被所谓尽最大努力交付。

然而，万一失败的全部或部分的事件总线，有可能性消息将丢失。

如果您的应用程序关心丢失的邮件，你应该代码您的处理程序是等幂的和你发件人可以在恢复后重试。

类型的消息

开箱即用 Vert.x 允许任何原始/简单类型、 字符串或buffers将作为邮件发送。

然而它是公约和惯例在 Vert.x 以json 格式发送邮件

JSON 是很容易创建、 读取和解析在 Vert.x 支持，因此它已成为一种通用语言为 Vert.x 的所有语言。

然而你不被迫使用 JSON，如果你不想去。

事件总线是非常灵活，也支持通过事件总线发送任意对象。你这样做是通过定义您想要发送的对象的codec。

事件总线 API

让我们跳进 API

获取事件总线

你获取到事件总线的引用，如下所示:

```
EventBus eb = vertx.eventBus();
```

还有每个 Vert.x 实例事件总线的单个实例。

注册处理程序

这个最简单的方法来注册一个处理程序用consumer。下面是一个示例:

```
EventBus eb = vertx.eventBus();

eb.consumer("news.uk.sport", message -> {
  System.out.println("I have received a message: " + message.body());
});
```

当邮件到达您的处理程序时，将调用您的处理程序，在message中传递.

从对 consumer() 的调用返回的对象是一个实例MessageConsumer 随后，此对象可以用于注销处理程序中，或使用处理程序以流的形式。

或者你可以使用consumer对返回 MessageConsumer 与没有处理程序设置，然后在那设置处理程序。例如:

```
EventBus eb = vertx.eventBus();

MessageConsumer<String> consumer = eb.consumer("news.uk.sport");
consumer.handler(message -> {
  System.out.println("I have received a message: " + message.body());
});
```

注册时聚集的事件总线上的一个处理程序，它可以有一些时间为要达到该群集的所有节点的登记。

如果你想要这已完成时通知，您可以注册的 MessageConsumer 对象上completion handler。

```
consumer.completionHandler(res -> {
  if (res.succeeded()) {
    System.out.println("The handler registration has reached all nodes");
  } else {
    System.out.println("Registration failed!");
  }
});
```

未注册的处理程序

若要撤消注册的处理程序，调用unregister.

如果你在一个群集事件总线，未登记可能需要一些时间来传播跨节点，如果你想要这完成后通知使用unregister.

```
consumer.unregister(res -> {
  if (res.succeeded()) {
    System.out.println("The handler un-registration has reached all nodes");
  } else {
    System.out.println("Un-registration failed!");
  }
});
```

发布消息

发布一条消息是简单的。只需使用publish指定地址发布到。

```
eventBus.publish("news.uk.sport", "Yay! Someone kicked a ball");
```

那消息然后将送交注册的反对地址 news.uk.sport 的所有处理程序。

发送消息

发送一条消息将导致只有一个Handlers在接收该消息的地址注册。这是点对点的消息传递模式。这个Handlers的选择是一个非严格的循环方式。

发送一条消息时，可以send

```
eventBus.send("news.uk.sport", "Yay! Someone kicked a ball");
```

消息设置headers

通过event bus发送的消息还可以包含头文件。

这可以通过发送或发布时提供DeliveryOptions指定:

```
DeliveryOptions options = new DeliveryOptions();
options.addHeader("some-header", "some-value");
eventBus.send("news.uk.sport", "Yay! Someone kicked a ball", options);
```

消息顺序

和发送顺序相同。

消息对象

在消息Handlers中接收的对象是一个Message。

该消息的body对应于发送或发布的对象。

消息标头有headers。

确认消息/发送答复

当使用send event bus 尝试将消息传递到event bus中注册的MessageConsumer 。

在某些情况下是有用的，发送者知道什么时候，消费者已经收到了消息和"processing"。

要确认该消息已被处理，消费者可以通过调用reply对消息进行回复。

当发生这种情况时，它会导致一个回复，发送回发送端和应答处理程序被调用的答复。

一个例子可以说明这一点：

接收:

```
MessageConsumer<String> consumer = eventBus.consumer("news.uk.sport");
consumer.handler(message -> {
  System.out.println("I have received a message: " + message.body());
  message.reply("how interesting!");
});
```

发送:

```
eventBus.send("news.uk.sport", "Yay! Someone kicked a ball across a patch of grass", a
r -> {
  if (ar.succeeded()) {
    System.out.println("Received reply: " + ar.result().body());
  }
});
```

应答可以包含一个消息体，它包含有用的信息。

"processing"实际上指的是应用程序定义和完全取决于在什么消息上消费者并不是 Vert.x event bus 本身知道或关心的东西。

一些例子:

- 一个简单的消息消费者，它实现了一个服务，它返回的时间是一天中的时间，将确认一个消息，包含时间的答复正文
- 如果消息被成功地保存在存储中，或不正确的话，它将实现一个持久队列的消息，如果消息被成功地保存，可能会对其进行应答。
- 如果消息成功处理，它可以从数据库中删除

发送超时

发送答复处理消息时你可以在DeliveryOptions中指定超时.

如果在该时间内没有收到答复，答复处理程序将调用失败。

默认的超时时间为 30 秒。

发送失败

消息发送失败有其他原因，包括:

- 没有可用来向其发送消息的handlers
- 收件人已明确地使用fail（失败）的消息

在所有情况下回复处理程序将调用具体的失败。

消息编解码器

如果你定义并注册了一个message codec，你可以将任何对象发送到event bus 上。

消息编解码器有一个名称，您在发送或发布该消息时在DeliveryOptions中指定该名称:

```
eventBus.registerCodec(myCodec);

DeliveryOptions options = new DeliveryOptions().setCodecName(myCodec.name());

eventBus.send("orders", new MyPOJO(), options);
```

如果你总是希望使用相同的编解码器，用于特定类型，那么你可以注册成为一个默认编码解码器，然后你不需要对每个发送的传递选项中指定编码解码器:

```
eventBus.registerDefaultCodec(MyPOJO.class, myCodec);

eventBus.send("orders", new MyPOJO());
```

注销消息编解码器使用unregisterCodec.

消息的编码解码器不总是要为相同的类型进行编码和解码。例如，您可以编写允许发送，MyPOJO 类的编解码器，但该消息发送到一个处理程序时它到达作为一个 MyOtherPOJO 类。

群集Event Bus

Event Bus并不仅仅存在于一个单一的Vert.x实例。通过网络上的不同集群实例Vert.x一起就可以形成一个单一的，分布式的，Event Bus。

集群编程

如果您要以编程方式创建您的 Vert.x 实例，通过将 Vert.x 实例配置获得集群的event bus;

```
VertxOptions options = new VertxOptions();
Vertx.clusteredVertx(options, res -> {
  if (res.succeeded()) {
    Vertx vertx = res.result();
    EventBus eventBus = vertx.eventBus();
    System.out.println("We now have a clustered event bus: " + eventBus);
  } else {
    System.out.println("Failed: " + res.cause());
  }
});
```

你还应该确保你已经在你的类路径中实现ClusterManager，例如默认 *HazelcastClusterManager*。

命令行上的集群

可以在命令行中运行 Vert.x 集群

```
vertx run my-verticle.js -cluster
```

自动清理 Verticles

如果你从 verticles里注册event bus处理程序，当verticle取消部署时处理程序将被自动注销。

JSON

不像一些其他语言，Java没有对JSON提供一流的支持，所以我们提供了两个类，来使你的应用程序Vert.x处理JSON更容易一点。

JSON 对象

JsonObject类表示 JSON 对象。

JSON对象基本上是有字符串键和值的map，值可以是JSON的一个支持的类型（字符串，数字，布尔值）。

JSON 对象还支持 null 值。

创建 **JSON** 对象

可以使用默认的构造函数创建空的 JSON 对象。

可以通过 JSON 格式字符串创建一个 JSON 对象，如下所示:

```
String jsonString = "{\"foo\":\"bar\"}";
JsonObject object = new JsonObject(jsonString);
```

条目放入 **JSON** 对象

使用put方法将值放入的 JSON 对象。

该方法可以链接调用，因为使用 fluent API:

```
JsonObject object = new JsonObject();
object.put("foo", "bar").put("num", 123).put("mybool", true);
```

从 **JSON** 对象中获取值

使用getXXX方法从 JSON 对象中获取值，例如：

```
String val = jsonObject.getString("some-key");
int intVal = jsonObject.getInteger("some-other-key");
```

JSON 对象转换为字符串

使用 encode 对象编码为一个字符串形式。

JSON 数组

JsonArray类表示 JSON 数组。

一个 JSON 数组是一个序列的值 (字符串、 数字、 布尔值)。

JSON 数组也可以包含空值。

创建 **JSON** 数组

可以使用默认的构造函数创建空的 JSON 数组。

可以通过 JSON 格式字符串创建一个 JSON 数组，如下所示:

```
String jsonString = "[\"foo\",\"bar\"]";
JsonArray array = new JsonArray(jsonString);
```

添加一个条目到 **JSON** 数组

使用add方法将条目添加到 JSON 数组。

```
JsonArray array = new JsonArray();
array.add("foo").add(123).add(false);
```

从 **JSON** 数组中获取值

使用getXXX方法从 JSON 数组中获取值，例如：

```
String val = array.getString(0);
Integer intVal = array.getInteger(1);
Boolean boolVal = array.getBoolean(2);
```

JSON 数组转换为字符串

使用encode方法将数组编码成字符串形式。

缓冲区

大多数数据是使用缓冲区抛入Vert.x内。

缓冲区是零个或多个字节，可以读取或写入，并自动扩展，以适应任何字节写入它。你或许可以认为缓冲区是智能字节数组。

创建缓冲区

缓冲区可以通过静态的Buffer.buffer方法创建。

可以从字符串或字节数组，初始化缓冲区，也可以创建空缓冲区。

下面是创建缓冲区的一些示例:

创建一个新的空缓冲区:

```
Buffer buff = Buffer.buffer();
```

从一个字符串创建一个缓冲区。字符串将在缓冲区中使用 utf-8 编码。

```
Buffer buff = Buffer.buffer("some string");
```

从字符串创建缓冲区：该字符串将使用指定的编码，例如：

```
Buffer buff = Buffer.buffer("some string", "UTF-16");
```

从 byte[]创建一个缓冲区。

```
byte[] bytes = new byte[] {1, 3, 5};
Buffer buff = Buffer.buffer(bytes);
```

创建一个初始大小的缓冲区。如果你知道你的缓冲区将有一定数量的数据写入，你可以在创建缓冲区时指定缓冲区大小。这使得缓冲最初分配多的内存，比缓冲区自动调整多次作为数据写入它更有效。

请注意，这种方法创建的缓冲区是空的。它不能创建大小为零的缓冲。

```
Buffer buff = Buffer.buffer(10000);
```

写入缓冲区

有两种方法来写入缓冲区：添加、和随机存取。在任何情况下，缓冲区将自动扩展，以包括字节。用缓冲区获取IndexOutOfBoundsException这是不可能的。

添加到缓冲区

要添加到缓冲区，您可以使用appendXXX方法。添加各种不同类型存在的方法。

appendXXX方法的返回值是缓冲区本身，以便这些可以将链接:

```
Buffer buff = Buffer.buffer();

buff.appendInt(123).appendString("hello\n");

socket.write(buff);
```

写入随机存取缓冲区

使用setXXX方法，在特定的索引中，可以写入缓冲区中。Set 的方法存在各种不同的数据类型。所有的设置的方法取索引作为第一个参数-这代表缓冲区中的位置从哪里开始写入数据。

缓冲区将总是需要时扩展，以容纳数据。

```
Buffer buff = Buffer.buffer();

buff.setInt(1000, 123);
buff.setString(0, "hello");
```

从缓冲区读取

使用getXXX方法从缓冲区读取数据。获得方法存在不同的数据类型。这些方法的第一个参数是获取数据的缓冲区中的一个索引。

```
Buffer buff = Buffer.buffer();
for (int i = 0; i < buff.length(); i += 4) {
  System.out.println("int value at " + i + " is " + buff.getInt(i));
}
```

使用无符号数字

无符号的数字可以通过getUnsignedXXX、 appendUnsignedXXX和setUnsignedXXX的方法读取或追加/套到缓冲区。这是有用的，当实现一个编解码器的网络协议优化，以尽量减少带宽消耗。

在以下示例中，200 是设置在指定的位置只有一个字节:

```
Buffer buff = Buffer.buffer(128);
int pos = 15;
buff.setUnsignedByte(pos, (short) 200);
System.out.println(buff.getUnsignedByte(pos));
```

控制台显示 '200'。

缓冲区长度

使用length来获取缓冲区的长度。缓冲区的长度是在缓冲区的最大索引 + 1 字节的索引。

复制缓冲区

使用copy，复制缓冲区

切片的缓冲区

切片的缓冲区是一个新的缓冲区，背对着原来的缓冲区，即它将不复制的基础数据。使用slice创建切片的缓冲区

缓冲区重新使用

一个缓冲区写入套接字或其它类似的地方之后，他们不能被重新使用。

编写 TCP 服务器和客户端

Vert.x 可以轻松地编写非阻塞的 TCP 客户端和服务器。

编写 TCP 服务器

创建 TCP 服务器

使用最简单的方法来创建一个 TCP 服务器，使用所有默认选项如下所示:

```
NetServer server = vertx.createNetServer();
```

配置 TCP 服务器

如果你不想默认值，可以将服务器配置通过传入一个NetServerOptions实例来创建它:

```
NetServerOptions options = new NetServerOptions().setPort(4321);
NetServer server = vertx.createNetServer(options);
```

启动服务器监听

使用listen告诉服务监听传入的请求

告诉要听的主机和端口作为选项中指定的服务器:

需要在选项（`NetServerOptions`）中指定的主机和端口:

```
NetServer server = vertx.createNetServer();
server.listen();
```

或在调用listen中指定的主机和端口，忽略`NetServerOptions`配置:

```
NetServer server = vertx.createNetServer();
server.listen(1234, "localhost");
```

默认主机是`0.0.0.0`，意味着 '监听所有可用的地址'，默认端口是0，这是一个特殊值，告诉服务器随机找一个未使用的本地端口使用。

眞实的绑定是异步的，所以服务器可能不会实际被侦听，直到有一段时间后，调用返回。

如果想要listen实际监听后通知你，可以提供listen调用处理程序。例如:

```
NetServer server = vertx.createNetServer();
server.listen(1234, "localhost", res -> {
  if (res.succeeded()) {
    System.out.println("Server is now listening!");
  } else {
    System.out.println("Failed to bind!");
  }
});
```

监听随机端口

如果 `0` 用作的监听端口，则服务器将找到一个未使用的随机端口监听。

若要找出服务器正在监听的真正端口，您可以调用actualPort.

```
NetServer server = vertx.createNetServer();
server.listen(0, "localhost", res -> {
  if (res.succeeded()) {
    System.out.println("Server is now listening on actual port: " + server.actualPort(
));
  } else {
    System.out.println("Failed to bind!");
  }
});
```

传入连接通知

若要连接时通知您需要设置connectHandler:

```
NetServer server = vertx.createNetServer();
server.connectHandler(socket -> {
  // Handle the connection in here
});
```

当进行连接时的处理程序将调用Netsocket实例。

这是一个类似于socket-like的接口的真实连接，并且允许你读写数据以及做其他各种类似的事情，比如关闭套接字。

从Socket读取数据

从socket读取数据要在socket上设置handler。

每次在socket上接收到数据Buffer实例，将调用此处理程序。

```
NetServer server = vertx.createNetServer();
server.connectHandler(socket -> {
  socket.handler(buffer -> {
    System.out.println("I received some bytes: " + buffer.length());
  });
});
```

数据写入socket

使用write写到socket.

```
Buffer buffer = Buffer.buffer().appendFloat(12.34f).appendInt(123);
socket.write(buffer);

// Write a string in UTF-8 encoding
socket.write("some data");

// Write a string using the specified encoding
socket.write("some data", "UTF-16");
```

写操作是异步的，调用返回之后可能不会发生。

关闭的处理程序（handler）

如果你想要关闭socket时得到通知，可以设置closeHandler：

```
socket.closeHandler(v -> {
  System.out.println("The socket has been closed");
});
```

处理异常

您可以设置exceptionHandler接收socket发生的任何异常。

Event bus写handler

每个socket自动注册event bus上的handler，当这个handler接收到任何buffers时，它会将它们写入到本身。

这使您可以将数据写到socket，它可能是在完全不同的verticle或甚至在不同的 Vert.x 实例，通过将buffers发送到该处理程序的地址。

由writeHandlerID给出了处理程序的地址

本地和远程地址

可以使用localAddress检索NetSocket的本地地址.

可以使用remoteAddress检索远程地址 (即地址连接的另一端) 的NetSocket.

从classpath发送文件或资源

直接使用sendFile就可以将文件写入socket。这是非常有效的发送文件的方法，因为它可以由操作系统内核直接支持。

```
socket.sendFile("myfile.dat");
```

Streaming sockets

NetSocket的实例也是ReadStream和WriteStream的实例，因此他们可以被用于泵送数据，可以从其他的读和写streams。

详细信息，请参阅Streams章。

升级到 SSL/TLS 连接

非 `SSL/TLS` 连接可以使用upgradeToSsl升级到 `SSL/TLS` 。

服务器或客户端必须配置为 `SSL/TLS` 才能正常工作。详细信息请参阅关于 `SSL/TLS` 章节。

关闭一个 TCP 服务器

叫close来关闭服务器。关闭服务器关闭任何打开的连接，并释放所有的服务器资源。

关闭是异步的，可能无法立即返回。如果你想要关闭已完成然后通知，可以通过handler做到。

当关闭已全面完成，然后将调用此处理程序。

```
server.close(res -> {
  if (res.succeeded()) {
    System.out.println("Server is now closed");
  } else {
    System.out.println("close failed");
  }
});
```

Verticles 自动清理

如果您正在从 `verticles` 内创建 `TCP` 服务器和客户端，`verticle` 取消部署时这些服务器和客户端将被自动关闭。

Scaling - sharing TCP 服务器

任何TCP服务器的处理器总是在相同的事件循环线程执行。

这意味着，如果在多核的服务器上运行，而你只部署一个实例，那么最多使用一个核心。

为了利用你的服务器更多的核心，你将需要部署服务器的多个实例。

您可以以编程方式在代码中实例化多个实例：

```
for (int i = 0; i < 10; i++) {
  NetServer server = vertx.createNetServer();
  server.connectHandler(socket -> {
    socket.handler(buffer -> {
      // Just echo back the data
      socket.write(buffer);
    });
  });
  server.listen(1234, "localhost");
}
```

或者，如果您正在使用verticles你可以简单地通过使用命令行选项 `-instances` 部署服务器verticle的多个实例：

```
vertx run com.mycompany.MyVerticle -instances 10
```

或以编程方式部署verticle

```
DeploymentOptions options = new DeploymentOptions().setInstances(10);
vertx.deployVerticle("com.mycompany.MyVerticle", options);
```

一旦你这样做，你会发现echo服务器的功能与以前的功能相同，但可以利用您在您的服务器上的所有核心，可以处理更多的工作。

在这一点上你可能会问自己' 你怎么能有多个服务器监听同一主机和端口?尝试部署多个实例，肯定会端口冲突?'

Vert.x 确实有点神奇，如:

当你在同一个主机上部署另一个服务器，作为一个现有的服务器，它实际上并没有尝试在同一个主机/端口上创建一个新的服务器。

相反它内部维护只是一台服务器，将连接通过循环的方式分配处理程序。

因此 Vert.x TCP 服务部署可以超过可用CPU内核，每个实例是单个线程。

编写 TCP 客户端

创建 TCP 客户端

最简单的方法来创建一个 TCP 客户端，使用默认选项如下所示:

```
NetClient client = vertx.createNetClient();
```

配置 TCP 客户端

如果你不想使用默认值，则创建TCP 客户端时，通过传入NetClientOptions实例可以配置:

```
NetClientOptions options = new NetClientOptions().setConnectTimeout(10000);
NetClient client = vertx.createNetClient(options);
```

建立连接

要连接到的服务器可以使用connect，指定端口和服务器地址和一个handler，handler包含 NetSocket 和成功或者失败的结果。

```
NetClientOptions options = new NetClientOptions().setConnectTimeout(10000);
NetClient client = vertx.createNetClient(options);
client.connect(4321, "localhost", res -> {
  if (res.succeeded()) {
    System.out.println("Connected!");
    NetSocket socket = res.result();
  } else {
    System.out.println("Failed to connect: " + res.cause().getMessage());
  }
});
```

配置自动重连

无法连接可以配置为自动重连。配置setReconnectInterval和setReconnectAttempts.

> 注意 目前 Vert.x 不会尝试重新连接，如果连接失败，重连和重连间隔仅适用于创建初始连接。

```
NetClientOptions options = new NetClientOptions().
    setReconnectAttempts(10).//重连次数
    setReconnectInterval(500)//重连间隔

NetClient client = vertx.createNetClient(options);
```

默认情况下，自动重连不开启。

配置服务器和客户端使用 SSL/TLS

TCP 客户端和服务器可以配置使用TLS（安全传输层协议）-TLS 的早期版本被称为 SSL。

无论使用 SSL/TLS服务器和客户端 Api 是相同的，是否启用，通过NetClientOptions 或NetServerOptions实例配置。

在服务器上启用 SSL/TLS

通过ssl启用 SSL/TLS.

默认是禁用的。

为服务器指定密钥/证书

SSL / TLS服务器通常提供证书给客户以验证他们的身份。

针对服务器的几种配置证书/密钥的方式:

第一种方法是通过指定 `Java` 密钥存储并包含证书和私钥的位置。

Java 密钥存储库可以使用JDK附带的实用程序keytool工具管理。 此外应提供密钥存储库的密码:

```
NetServerOptions options = new NetServerOptions().setSsl(true).setKeyStoreOptions(
    new JksOptions().
        setPath("/path/to/your/server-keystore.jks").
        setPassword("password-of-your-keystore")
);
NetServer server = vertx.createNetServer(options);
```

或者，你可以把你自己的钥匙储存为一个缓冲区，直接提供：

```
Buffer myKeyStoreAsABuffer = vertx.fileSystem().readFileBlocking("/path/to/your/server
-keystore.jks");
JksOptions jksOptions = new JksOptions().
    setValue(myKeyStoreAsABuffer).
    setPassword("password-of-your-keystore");
NetServerOptions options = new NetServerOptions().
    setSsl(true).
    setKeyStoreOptions(jksOptions);
NetServer server = vertx.createNetServer(options);
```

还可以以类似 `JKS` 方式 `PKCS #12` 格式 (http://en.wikipedia.org/wiki/PKCS_12)密钥存储加载密钥/证书，通常具有.pfx或.p12扩展名:

```
NetServerOptions options = new NetServerOptions().setSsl(true).setPfxKeyCertOptions(
    new PfxOptions().
        setPath("/path/to/your/server-keystore.pfx").
        setPassword("password-of-your-keystore")
);
NetServer server = vertx.createNetServer(options);
```

此外支持缓冲区配置:

```
Buffer myKeyStoreAsABuffer = vertx.fileSystem().readFileBlocking("/path/to/your/server
-keystore.pfx");
PfxOptions pfxOptions = new PfxOptions().
    setValue(myKeyStoreAsABuffer).
    setPassword("password-of-your-keystore");
NetServerOptions options = new NetServerOptions().
    setSsl(true).
    setPfxKeyCertOptions(pfxOptions);
NetServer server = vertx.createNetServer(options);
```

另一种方式，分别使用 `.pem` 文件提供服务器私钥和证书。

```
NetServerOptions options = new NetServerOptions().setSsl(true).setPemKeyCertOptions(
    new PemKeyCertOptions().
        setKeyPath("/path/to/your/server-key.pem").
        setCertPath("/path/to/your/server-cert.pem")
);
NetServer server = vertx.createNetServer(options);
```

此外支持缓冲区配置:

```
Buffer myKeyAsABuffer = vertx.fileSystem().readFileBlocking("/path/to/your/server-key.
pem");
Buffer myCertAsABuffer = vertx.fileSystem().readFileBlocking("/path/to/your/server-cer
t.pem");
PemKeyCertOptions pemOptions = new PemKeyCertOptions().
    setKeyValue(myKeyAsABuffer).
    setCertValue(myCertAsABuffer);
NetServerOptions options = new NetServerOptions().
    setSsl(true).
    setPemKeyCertOptions(pemOptions);
NetServer server = vertx.createNetServer(options);
```

请牢记，pem 的配置， 私钥是不加密的。

指定信任服务器

SSL / TLS服务器使用证书授权可以以验证客户端的身份。

几种针对服务器的证书授权的配置方法:

此外应提供密钥存储库的密码:

Java trust存储库可以使用 JDK附带的实用程序keytool工具管理。

此外应提供trust存储库的密码:

```
NetServerOptions options = new NetServerOptions().
    setSsl(true).
    setClientAuth(ClientAuth.REQUIRED).
    setTrustStoreOptions(
        new JksOptions().
            setPath("/path/to/your/truststore.jks").
            setPassword("password-of-your-truststore")
    );
NetServer server = vertx.createNetServer(options);
```

或者，你可以把你自己的trust储存为一个缓冲区，直接提供：

```
Buffer myTrustStoreAsABuffer = vertx.fileSystem().readFileBlocking("/path/to/your/trus
tstore.jks");
NetServerOptions options = new NetServerOptions().
    setSsl(true).
    setClientAuth(ClientAuth.REQUIRED).
    setTrustStoreOptions(
        new JksOptions().
            setValue(myTrustStoreAsABuffer).
            setPassword("password-of-your-truststore")
    );
NetServer server = vertx.createNetServer(options);
```

还可以以类似 `JKS` 方式 `PKCS #12` 格式 (http://en.wikipedia.org/wiki/PKCS_12) `trust` 存储加载密钥/证书，通常具有.pfx或.p12扩展名:

```
NetServerOptions options = new NetServerOptions().
    setSsl(true).
    setClientAuth(ClientAuth.REQUIRED).
    setPfxTrustOptions(
        new PfxOptions().
            setPath("/path/to/your/truststore.pfx").
            setPassword("password-of-your-truststore")
    );
NetServer server = vertx.createNetServer(options);
```

此外支持缓冲区配置:

```
Buffer myTrustStoreAsABuffer = vertx.fileSystem().readFileBlocking("/path/to/your/trus
tstore.pfx");
NetServerOptions options = new NetServerOptions().
    setSsl(true).
    setClientAuth(ClientAuth.REQUIRED).
    setPfxTrustOptions(
        new PfxOptions().
            setValue(myTrustStoreAsABuffer).
            setPassword("password-of-your-truststore")
    );
NetServer server = vertx.createNetServer(options);
```

另一种方式，使用列表的.pem文件提供服务器证书授权

```
NetServerOptions options = new NetServerOptions().
    setSsl(true).
    setClientAuth(ClientAuth.REQUIRED).
    setPemTrustOptions(
        new PemTrustOptions().
            addCertPath("/path/to/your/server-ca.pem")
    );
NetServer server = vertx.createNetServer(options);
```

此外支持缓冲区配置:

```
Buffer myCaAsABuffer = vertx.fileSystem().readFileBlocking("/path/to/your/server-ca.pf
x");
NetServerOptions options = new NetServerOptions().
    setSsl(true).
    setClientAuth(ClientAuth.REQUIRED).
    setPemTrustOptions(
        new PemTrustOptions().
            addCertValue(myCaAsABuffer)
    );
NetServer server = vertx.createNetServer(options);
```

在客户端上启用 **SSL/TLS**

Net 客户端也可以轻松地配置为使用 SSL。他们有相同的 API，当使用 SSL时 使用的是标准的sockets。

NetClient调用setSSL(true) 函数启用 SSL。

客户端信任配置

在客户端，如果trustALl设置为 true，客户端将信任所有的服务器证书。连接仍将被加密，但这种模式是易受攻击的。请谨慎使用。默认值为 false。

```
NetClientOptions options = new NetClientOptions().
    setSsl(true).
    setTrustAll(true);
NetClient client = vertx.createNetClient(options);
```

如果未设置trustAll，客户端必须配置trust store 和应该包含该客户端信任的服务器证书。

同服务器配置，配置客户端信任可以分几个方面:

第一种方法是通过指定包含证书授权 Java trust-store 的位置。

这只是标准 Java 密钥库，密钥存储与服务器端相同。由 jks options 使用函数path设置的客户端trust-store位置。如果服务器不是客户端信任存储区中的连接过程中提供的证书，则连接尝试不会成功。

```
NetClientOptions options = new NetClientOptions().
    setSsl(true).
    setTrustStoreOptions(
        new JksOptions().
            setPath("/path/to/your/truststore.jks").
            setPassword("password-of-your-truststore")
    );
NetClient client = vertx.createNetClient(options);
```

此外支持缓冲区配置:

```
Buffer myTrustStoreAsABuffer = vertx.fileSystem().readFileBlocking("/path/to/your/trus
tstore.jks");
NetClientOptions options = new NetClientOptions().
    setSsl(true).
    setTrustStoreOptions(
        new JksOptions().
            setValue(myTrustStoreAsABuffer).
            setPassword("password-of-your-truststore")
    );
NetClient client = vertx.createNetClient(options);
```

还可以以类似 `JKS` 方式 `PKCS #12` 格式 (http://en.wikipedia.org/wiki/PKCS_12) `trust` 存储加载密钥/证书，通常具有.pfx或.p12扩展名:

```
NetClientOptions options = new NetClientOptions().
    setSsl(true).
    setPfxTrustOptions(
        new PfxOptions().
            setPath("/path/to/your/truststore.pfx").
            setPassword("password-of-your-truststore")
    );
NetClient client = vertx.createNetClient(options);
```

此外支持缓冲区配置:

```
Buffer myTrustStoreAsABuffer = vertx.fileSystem().readFileBlocking("/path/to/your/trus
tstore.pfx");
NetClientOptions options = new NetClientOptions().
    setSsl(true).
    setPfxTrustOptions(
        new PfxOptions().
            setValue(myTrustStoreAsABuffer).
            setPassword("password-of-your-truststore")
    );
NetClient client = vertx.createNetClient(options);
```

另一种方式，使用列表的.pem文件提供服务器证书授权

```
NetClientOptions options = new NetClientOptions().
    setSsl(true).
    setPemTrustOptions(
        new PemTrustOptions().
            addCertPath("/path/to/your/ca-cert.pem")
    );
NetClient client = vertx.createNetClient(options);
```

此外支持缓冲区配置:

```
Buffer myTrustStoreAsABuffer = vertx.fileSystem().readFileBlocking("/path/to/your/ca-c
ert.pem");
NetClientOptions options = new NetClientOptions().
    setSsl(true).
    setPemTrustOptions(
        new PemTrustOptions().
            addCertValue(myTrustStoreAsABuffer)
    );
NetClient client = vertx.createNetClient(options);
```

为客户端指定密钥/证书

如果服务器要求客户端身份验证，然后连接时，客户端必须向服务器提交它自己的证书。客户端可以配置几个方面:

第一种方法是通过指定 Java 密钥库包含密钥和证书的位置。这是只是常规 Java 的密钥存储库。客户端密钥库位置是通过使用函数path上jks options设置.

```
NetClientOptions options = new NetClientOptions().setSsl(true).setKeyStoreOptions(
    new JksOptions().
        setPath("/path/to/your/client-keystore.jks").
        setPassword("password-of-your-keystore")
);
NetClient client = vertx.createNetClient(options);
```

此外支持缓冲区配置:

```
Buffer myKeyStoreAsABuffer = vertx.fileSystem().readFileBlocking("/path/to/your/client
-keystore.jks");
JksOptions jksOptions = new JksOptions().
    setValue(myKeyStoreAsABuffer).
    setPassword("password-of-your-keystore");
NetClientOptions options = new NetClientOptions().
    setSsl(true).
    setKeyStoreOptions(jksOptions);
NetClient client = vertx.createNetClient(options);
```

还可以以类似 `JKS` 方式 `PKCS #12` 格式 (http://en.wikipedia.org/wiki/PKCS_12) `trust` 存储加载密钥/证书，通常具有.pfx或.p12扩展名:

```
NetClientOptions options = new NetClientOptions().setSsl(true).setPfxKeyCertOptions(
    new PfxOptions().
        setPath("/path/to/your/client-keystore.pfx").
        setPassword("password-of-your-keystore")
);
NetClient client = vertx.createNetClient(options);
```

此外支持缓冲区配置:

```
Buffer myKeyStoreAsABuffer = vertx.fileSystem().readFileBlocking("/path/to/your/client
-keystore.pfx");
PfxOptions pfxOptions = new PfxOptions().
    setValue(myKeyStoreAsABuffer).
    setPassword("password-of-your-keystore");
NetClientOptions options = new NetClientOptions().
    setSsl(true).
    setPfxKeyCertOptions(pfxOptions);
NetClient client = vertx.createNetClient(options);
```

另一种方式，分别使用 `.pem` 文件提供服务器私钥和证书。

```
NetClientOptions options = new NetClientOptions().setSsl(true).setPemKeyCertOptions(
    new PemKeyCertOptions().
        setKeyPath("/path/to/your/client-key.pem").
        setCertPath("/path/to/your/client-cert.pem")
);
NetClient client = vertx.createNetClient(options);
```

此外支持缓冲区配置:

```
Buffer myKeyAsABuffer = vertx.fileSystem().readFileBlocking("/path/to/your/client-key.
pem");
Buffer myCertAsABuffer = vertx.fileSystem().readFileBlocking("/path/to/your/client-cer
t.pem");
PemKeyCertOptions pemOptions = new PemKeyCertOptions().
    setKeyValue(myKeyAsABuffer).
    setCertValue(myCertAsABuffer);
NetClientOptions options = new NetClientOptions().
    setSsl(true).
    setPemKeyCertOptions(pemOptions);
NetClient client = vertx.createNetClient(options);
```

请牢记，pem 的配置， 私钥是不加密的。

吊销认证授权

被吊销的证书不再应信任，信任可以配置为使用证书吊销列表 (CRL)。crlPath配置要使用的 crl 列表:

```
NetClientOptions options = new NetClientOptions().
    setSsl(true).
    setTrustStoreOptions(trustOptions).
    addCrlPath("/path/to/your/crl.pem");
NetClient client = vertx.createNetClient(options);
```

此外支持缓冲区配置:

```
Buffer myCrlAsABuffer = vertx.fileSystem().readFileBlocking("/path/to/your/crl.pem");
NetClientOptions options = new NetClientOptions().
    setSsl(true).
    setTrustStoreOptions(trustOptions).
    addCrlValue(myCrlAsABuffer);
NetClient client = vertx.createNetClient(options);
```

配置加密套件

默认情况下，TLS配置将使用JVM运行Vert.x.的加密套件这种密码套件可以用一组启用密码进行配置：

```
NetServerOptions options = new NetServerOptions().
    setSsl(true).
    setKeyStoreOptions(keyStoreOptions).
    addEnabledCipherSuite("ECDHE-RSA-AES128-GCM-SHA256").
    addEnabledCipherSuite("ECDHE-ECDSA-AES128-GCM-SHA256").
    addEnabledCipherSuite("ECDHE-RSA-AES256-GCM-SHA384").
    addEnabledCipherSuite("CDHE-ECDSA-AES256-GCM-SHA384");
NetServer server = vertx.createNetServer(options);
```

加密套件可以在NetServerOptions或NetClientOptions配置中指定。

编写 HTTP 服务器和客户端

Vert.x 允许您轻松地编写非阻塞的 HTTP 客户端和服务器。

编写 HTTP 服务器

创建一个 HTTP 服务器

最简单的方法来创建一个 HTTP 服务器，所有选项使用默认的。如下所示:

```
HttpServer server = vertx.createHttpServer();
```

配置 HTTP 服务器

如果你不想使用默认值，创建服务器时可以通过传入一个HttpServerOptions实例配置:

```
HttpServerOptions options = new HttpServerOptions().setMaxWebsocketFrameSize(1000000);

HttpServer server = vertx.createHttpServer(options);
```

启动服务器监听

使用listen告诉服务器以监听传入的请求。

在选项中指定的主机和端口：

```
HttpServer server = vertx.createHttpServer();
server.listen();
```

或在调用listen中指定的主机和端口，忽略配置选项:

```
HttpServer server = vertx.createHttpServer();
server.listen(8080, "myhost.com");
```

默认主机是 `0.0.0.0` ，'监听所有可用的地址'，默认端口是80.

实际的绑定是异步的，所以服务器可能不会实际被监听，直到一段时间后，调用返回。

如果想要listen实际监听后通知你，可以提供listen调用处理程序。例如:

```
HttpServer server = vertx.createHttpServer();
server.listen(8080, "myhost.com", res -> {
  if (res.succeeded()) {
    System.out.println("Server is now listening!");
  } else {
    System.out.println("Failed to bind!");
  }
});
```

收到传入请求通知

若要请求到达时通知，需要设置requestHandler:

```
HttpServer server = vertx.createHttpServer();
server.requestHandler(request -> {
  // Handle the request in here
});
```

处理请求

当请求到达时，则该请求调用处理程序传递HttpServerRequest的一个实例。此对象所表示的服务器端的 HTTP 请求。

当请求headers已完全读取时，将调用该处理程序。

如果该请求包含一个body，该body将到达服务器，一段时间后请求处理程序被调用。

服务器请求对象可以获取uri、path、 params和headers等。

每个服务器请求对象是与一台服务器的响应对象相关联。使用response来获取对HttpServerResponse对象。

这里是一个简单的例子，服务器处理请求和回复"hello world"。

```
vertx.createHttpServer().requestHandler(request -> {
  request.response().end("Hello world");
}).listen(8080);
```

请求的版本

HTTP 请求中指定的版本，可以用version获取

请求方法

method用于获得请求的 `HTTP` 方法 。(即是GET，POST、 PUT、 DELETE、 HEAD、OPTIONS等)。

请求的 **URI**

使用uri来获取请求的 URI。

注意，这是通过在 HTTP 请求中，实际 URI，它几乎总是相对 URI。

URI是在 HTTP规范的5.1.2节中定义 - 请求URI

请求路径

使用path返回 URI 的路径

例如，如果请求 URI 只是:

`a/b/c/page.html?param1=abc&param2=xyz`

那么，路径会

`/a/b/c/page.html`

请求查询

使用query返回的 URI 的查询部分

例如，如果请求 URI 只是:

`a/b/c/page.html?param1=abc&param2=xyz`

那么，该查询会

`param1=abc&param2=xyz`

请求**headers**

使用headers方法来返回的 HTTP 请求headers。

这将返回一个实例MultiMap-就像一个普通的map或Hash，但允许多个值的同一键-这是因为 HTTP 允许多个header值用相同的密钥。

`key` 不区分大小写，这就意味着您可以执行以下操作:

```
MultiMap headers = request.headers();

// Get the User-Agent:
System.out.println("User agent is " + headers.get("user-agent"));

// You can also do this and get the same result:
System.out.println("User agent is " + headers.get("User-Agent"));
```

请求参数

使用params返回的 HTTP 请求参数。

就像headers这返回的MultiMap实例，可以有多个参数具有相同的名称。

路径后的请求 URI是请求参数。例如，如果 URI:

```
/page.html?param1=abc&param2=xyz
```

然后参数将包含以下内容:

```
param1: 'abc'
param2: 'xyz
```

请注意这些请求参数从请求的 URL。如果您有已作为体内的multi-part/form-data请求提交 HTML 表单提交的一部分发送的窗体属性然后他们将不出现在这里的 params。

请注意，这些请求参数从请求的URL中获取。如果你的form属性为 `multi-part/form-data` 请求的话，参数不会出现在这里，会包含在body中提交发送。

远程地址

请求的发送者的地址可以通过remoteaddress获取。

绝对 **URI**

传入的 HTTP 请求的 URI 是通常相对。如果想要检索对应于该请求的绝对 URI，可以用absoluteURI

结束处理程序

当整个，包括任何body已完全读取请求时，将调用endHandler请求。

从请求正**body**读取数据

通常一个 HTTP 请求包含我们想要读取的body。前面所提到的请求处理程序仅仅有headers，这里并没有body。

这是因为body可能会非常大 (例如一个文件上传)，我们通常不不会吧缓冲的整个body放在内存中交给你，因为那将导致服务器内存用尽。

若要接收body，您可以使用handler请求，调用后，会有请求body到达。下面是一个示例:

```
request.handler(buffer -> {
  System.out.println("I have received a chunk of the body of length " + buffer.length(
));
});
```

传递到handler的对象是一个Buffer，该handler可以调用多次，当数据到达时从网络，根据body的大小。

在某些情况下 (例如如果body很小) 想要在内存中缓存整个body如下所示:

```
Buffer totalBuffer = Buffer.buffer();

request.handler(buffer -> {
  System.out.println("I have received a chunk of the body of length " + buffer.length(
));
  totalBuffer.appendBuffer(buffer);
});

request.endHandler(v -> {
  System.out.println("Full body received, length = " + totalBuffer.length());
});
```

这是这种常见的情况，Vert.x 提供bodyHandler来为你做这个。bodyHandler收到所有的body:

```
request.bodyHandler(totalBuffer -> {
  System.out.println("Full body received, length = " + totalBuffer.length());
});
```

Pumping requests

请求对象是ReadStream ，所以你可以pump请求body到任何WriteStream实例。

处理 HTML 表单

HTML表单可以提交 `application/x-www-form-urlencoded` 或 `multipart/form-data` 内容类型。

对于 url 编码形式，像正常的查询参数一样，把表单属性编码在 url 中。

multi-part表单在请求body中编码，因此不可用直到从wire读取了整个body。

Multi-part 表单还可以包含文件上传。

如果您想要获取的属性是multi-part的形式，得到这种表单之前，通过调用setExpectMultipart设置为true，告诉 Vert.x 你期望的body是read，，然后你应该使用formAttributes获取 ，读取所有body的属性:

```
server.requestHandler(request -> {
  request.setExpectMultipart(true);
  request.endHandler(v -> {
    // The body has now been fully read, so retrieve the form attributes
    MultiMap formAttributes = request.formAttributes();
  });
});
```

处理表单文件上传

Vert.x 还可以处理在multi-part请求body中编码文件上传。

接收文件上传，告诉 Vert.x 期望multi-part表单形式并要求设置uploadHandler。

每个上传到服务器上，此handler将调用一次。

传递到handler的对象是一个HttpServerFileUpload实例。

```
server.requestHandler(request -> {
  request.setExpectMultipart(true);
  request.uploadHandler(upload -> {
    System.out.println("Got a file upload " + upload.name());
  });
});
```

上传的文件可能会很大，我们不提供单个缓冲区上传整个数据，因为这可能导致内存消耗殆尽，取而代之的是，上传的数据在块中收到：

```
request.uploadHandler(upload -> {
  upload.handler(chunk -> {
    System.out.println("Received a chunk of the upload of length " + chunk.length());
  });
});
```

上传对象是ReadStream ，所以你可以pump请求body到任何WriteStream实例。

如果你只是想要上传文件到磁盘，你可以使用streamToFileSystem:

```
request.uploadHandler(upload -> {
  upload.streamToFileSystem("myuploads_directory/" + upload.filename());
});
```

> 警告 确保您在一个生产系统中检查文件，以避免恶意客户端将文件上传到您的文件系统。查看更多信息的安全说明。

发送返回响应

服务器的响应对象是HttpServerResponse的实例和所得的请求response.

响应对象用于写回 HTTP 客户端的响应。

设置状态代码和消息

response 的默认 HTTP 状态码是200，表示OK.

使用setStatusCode来设置不同的代码。

您还可以使用setStatusMessage指定一个自定义的状态消息.

如果您不指定一条状态消息，将使用默认的状态代码。

写入 **HTTP responses**（响应）

使用一个write操作，将数据写入 HTTP responses。

这些可以在响应结束之前多次调用。可以以下面几种方法调用:

用一个缓冲区:

```
HttpServerResponse response = request.response();
response.write(buffer);
```

用一个字符串。在这种情况下将默认使用UTF-8编码。

```
HttpServerResponse response = request.response();
response.write("hello world!");
```

使用一个字符串和编码。在本例的字符串将使用指定的编码。

```
HttpServerResponse response = request.response();
response.write("hello world!", "UTF-16");
```

写入responses是异步的，在写入队列后立即返回。

如果你只是写一个字符串或缓存HTTP response,你可以写它和调用end结束response

在响应头第一个调用正在被写入结果的响应 第一次调用将写入结果正在被写入到响应响应头。因此，如果您不使用 HTTP 分块然后之前，必须设置Content-Length标头写的反应，因为它否则就太晚了。如果您使用的 HTTP 分块你不必担心。

第一次调用写结果的响应头被写入响应。因此,如果你不使用HTTP分块，那么header，你必须设置响应的Content-Length。如果您使用HTTP分块你不必担心。

结束 HTTP responses

一旦你完成了 HTTP responses你应该end它。

这可以有以下几种方式:

不带任何参数，response 是只是结束。

```
HttpServerResponse response = request.response();
response.write("hello world!");
response.end();
```

它也可以用字符串调用或和缓冲区相同的方式 write。在这种情况下，首先用字符串写入缓冲区，然后和不带参数的一样。例如：

```
HttpServerResponse response = request.response();
response.end("hello world!");
```

关闭底层连接

使用 close 关闭底层的 TCP 连接.

短连接在 response 结束时， `Vert.x` 将自动关闭 。

长连接的 Vert.x 默认情况下是不会自动的关闭。如果你希望连接保持到空闲时间后关闭，使用 setIdleTimeout 方法配置。

设置 response headers

可以将 HTTP response headers 通过 response 直接添加到 headers:

```
HttpServerResponse response = request.response();
MultiMap headers = response.headers();
headers.set("content-type", "text/html");
headers.set("other-header", "wibble");
```

或者你可以使用putHeader

```
HttpServerResponse response = request.response();
response.putHeader("content-type", "text/html").putHeader("other-header", "wibble");
```

Headers 必须添加在所有写入response body之前。

分块 **HTTP** 响应和传输

Vert.x 支持HTTP 分块传输编码.

表示输出的内容长度不能确定。

把 HTTP response 设置为分块模式，如下所示:

```
HttpServerResponse response = request.response();
response.setChunked(true);
```

默认值为非分块。在分块模式下，每次调用 write 调用时，就会写出一个新的 HTTP 块。

当在分块模式中你也可以写入 HTTP response 传输到response。实际上，这些实际上写入响应的最后块。

若要添加trailers到response，直接添加到trailers.

```
HttpServerResponse response = request.response();
response.setChunked(true);
MultiMap trailers = response.trailers();
trailers.set("X-wibble", "woobble").set("X-quux", "flooble");
```

或使用putTrailer.

```
HttpServerResponse response = request.response();
response.setChunked(true);
response.putTrailer("X-wibble", "woobble").putTrailer("X-quux", "flooble");
```

直接从磁盘或类路径（**classpath**）提供文件服务

如果你正在编写一个Web服务器，使用AsyncFile打开磁盘上的文件并注入到HTTP response。

或者你可以一气呵成，使用readFile加载并直接写入响应。

另外，Vert.x 提供了一种方法，可以从磁盘或者文件系统操作HTTP response。由底层操作系统支持，这可能在OS中直接从文件到套接字传送字节，而无需通过用户空间（user-space）复制。

对于大文件使用sendFile，通常更有效，但小文件可能较慢。

下面是使用sendFile，提供了一个非常简单的Web服务器:

```
vertx.createHttpServer().requestHandler(request -> {
  String file = "";
  if (request.path().equals("/")) {
    file = "index.html";
  } else if (!request.path().contains("..")) {
    file = request.path();
  }
  request.response().sendFile("web/" + file);
}).listen(8080);
```

发送的文件是异步的,可能无法马上返回。如果你想要该文件写入完成通知，可以使用sendFile

注意

> 如果你在使用的 HTTPS 使用sendFile会通过用户空间 ，因为如果内核将数据直接从磁盘对套接字,没有机会给应用任何加密。

警告

> 如果你要直接使用 Vert.x 写 web 服务器，小心用户访问其他文件路径，使用 Vert.x Web 可能更安全，。

当只需要一个文件片段，可以给定从某字节开发，可以通过下面达到：

```
vertx.createHttpServer().requestHandler(request -> {
  long offset = 0;
  try {
    offset = Long.parseLong(request.getParam("start"));
  } catch (NumberFormatException e) {
    // error handling...
  }

  long end = Long.MAX_VALUE;
  try {
    end = Long.parseLong(request.getParam("end"));
  } catch (NumberFormatException e) {
    // error handling...
  }

  request.response().sendFile("web/mybigfile.txt", offset, end);
}).listen(8080);
```

如果想要发送的文件从偏移量开始到结束，您不需要提供长度，在这种情况下你可以这么做:

```
vertx.createHttpServer().requestHandler(request -> {
  long offset = 0;
  try {
    offset = Long.parseLong(request.getParam("start"));
  } catch (NumberFormatException e) {
    // error handling...
  }

  request.response().sendFile("web/mybigfile.txt", offset);
}).listen(8080);
```

注入responses

服务器response是一个WriteStream实例，可以从任何ReadStream注入，例如AsyncFile，NetSocket，WebSocket或HttpServerRequest。

这是一个PUT请求body返回响应的例子，使用pump，即使HTTP请求body比内存大的多，也能正常工作：

```
vertx.createHttpServer().requestHandler(request -> {
  HttpServerResponse response = request.response();
  if (request.method() == HttpMethod.PUT) {
    response.setChunked(true);
    Pump.pump(request, response).start();
    request.endHandler(v -> response.end());
  } else {
    response.setStatusCode(400).end();
  }
}).listen(8080);
```

HTTP 压缩

Vert.x 带有 HTTP 压缩开箱即用支持。

这意味着在发送回客户端之前能自动压缩响应的主体。

如果客户端不支持 HTTP 压缩，将没有压缩 body 响应发送回。

这可以同时处理支持 HTTP 压缩的客户端和那些不支持HTTP 压缩的客户端。

若要启用压缩，使用setCompressionSupported配置。

默认情况下不启用压缩。

当启用 HTTP 压缩，服务器将检查客户端是否包括Accept-Encoding的报头，它包含支持的压缩方式。常用的有deflate和 gzip(Web服务器处理HTTP压缩之gzip、deflate压缩)。 Vert.x 两者都支持。

如果服务器将自动压缩与一个支持压缩响应正文中的并将其发送回客户端，就找到这种头。如果找到这样的报头，服务器将自动压缩发送回客户端。

要知道压缩可能能够减少网络流量，但是是需要消耗更多的 `CPU` 。

编写 HTTP 客户端

创建 HTTP 客户端

使用默认选项创建一个HttpClient实例，如下所示:

```
HttpClient client = vertx.createHttpClient();
```

如果您想要在创建时配置客户端的选项，如下所示:

```
HttpClientOptions options = new HttpClientOptions().setKeepAlive(false);
HttpClient client = vertx.createHttpClient(options);
```

发出请求

Http 客户端非常灵活，可以用各种方式请求。

http 客户端经常需要将许多请求发送到相同的主机/端口 。为了避免每次发出请求需要重复配置主机/端口，您可以配置客户端的默认主机/端口:

```
HttpClientOptions options = new HttpClientOptions().setDefaultHost("wibble.com");
// Can also set default port if you want...
HttpClient client = vertx.createHttpClient(options);
client.getNow("/some-uri", response -> {
  System.out.println("Received response with status code " + response.statusCode());
});
```

或者可以指定单个请求的主机/端口。

```
HttpClient client = vertx.createHttpClient();

// Specify both port and host name
client.getNow(8080, "myserver.mycompany.com", "/some-uri", response -> {
  System.out.println("Received response with status code " + response.statusCode());
});

// This time use the default port 80 but specify the host name
client.getNow("foo.othercompany.com", "/other-uri", response -> {
  System.out.println("Received response with status code " + response.statusCode());
});
```

没有请求body简单的请求

通常情况下，你会想要与没有请求body的 HTTP 请求。这通常是 HTTP GET，OPTIONS和HEAD请求。

这是 Vert.x http 客户端的最简单方法，使用前缀与Now的方法。例如getNow。

这些方法创建HTTP请求，并在单个方法调用发送，让你提供一个处理程序，将与HTTP响应时调用它回来。

```
HttpClient client = vertx.createHttpClient();

// Send a GET request
client.getNow("/some-uri", response -> {
  System.out.println("Received response with status code " + response.statusCode());
});

// Send a GET request
client.headNow("/other-uri", response -> {
  System.out.println("Received response with status code " + response.statusCode());
});
```

编写普通的**requests**

不知道想要发送请求方法，运行时才知道。这个用例我们提供通用请求方法例如request，可以在运行时指定 HTTP 方法:

```
HttpClient client = vertx.createHttpClient();

client.request(HttpMethod.GET, "some-uri", response -> {
  System.out.println("Received response with status code " + response.statusCode());
}).end();

client.request(HttpMethod.POST, "foo-uri", response -> {
  System.out.println("Received response with status code " + response.statusCode());
}).end("some-data");
```

编写请求主体（**bodies**）

有时想要在requests里含有body，或者想写请求的headers。

可以使用post方法，或者普通的request方法

这些方法发送的请求不会立即返回，而是返回HttpClientRequest实例，用于写入body和header。

下面是一些写入body和header的 POST 请求的例子:

```
HttpClient client = vertx.createHttpClient();

HttpClientRequest request = client.post("some-uri", response -> {
  System.out.println("Received response with status code " + response.statusCode());
});

// Now do stuff with the request
request.putHeader("content-length", "1000");
request.putHeader("content-type", "text/plain");
request.write(body);

// Make sure the request is ended when you're done with it
request.end();

// Or fluently:

client.post("some-uri", response -> {
  System.out.println("Received response with status code " + response.statusCode());
}).putHeader("content-length", "1000").putHeader("content-type", "text/plain").write(b
ody).end();

// Or event more simply:

client.post("some-uri", response -> {
  System.out.println("Received response with status code " + response.statusCode());
}).putHeader("content-type", "text/plain").end(body);
```

默认utf-8编码：

```
request.write("some data");

// Write string encoded in specific encoding
request.write("some other data", "UTF-16");

// Write a buffer
Buffer buffer = Buffer.buffer();
buffer.appendInt(123).appendLong(245l);
request.write(buffer);
```

如果你的 HTTP 请求只写入单个字符串或缓冲区，你可以写它和结束end函数单一调用中的请求。

```
request.end("some simple data");

// Write buffer and end the request (send it) in a single call
Buffer buffer = Buffer.buffer().appendDouble(12.34d).appendLong(432l);
request.end(buffer);
```

当你写到一个请求时，write第一次调用会导致写入网络的请求标头。

实际的写操作是异步的之前一段时间后调用已返回, 不是可能发生。

非分块请求正文与 HTTP 请求需要要提供的Content-Length标头。

因此，如果您不使用分块的 HTTP 然后之前，必须设置Content-Length标头写入请求，否则它将太迟了。

如果您正在调用接受字符串或缓冲区的end方法之一然后 Vert.x 将自动计算并设置Content-Length标头在写请求正文之前。

如果您使用的 HTTP 分块Content-Length标头不是必需的所以你不需要计算的前期的大小。